Consultative Committee for Space Data Systems

RECOMMENDATION FOR SPACE DATA SYSTEMS STANDARDS

ADVANCED ORBITING SYSTEMS, NETWORKS AND DATA LINKS:

FORMAL SPECIFICATION OF THE PATH SERVICE AND PROTOCOL

ADDENDUM TO CCSDS 701.0-B-2

CCSDS 705.2-B-1

BLUE BOOK

May 1994

AUTHORITY

Issue:	Blue Book, Issue 1
Date:	May 1994
Location:	Villafranca, Spain

This Recommendation is published and maintained by:

CCSDS Secretariat
Program Integration Division (Code OI)
National Aeronautics and Space Administration
Washington, DC 20546, USA

STATEMENT OF INTENT

The Consultative Committee for Space Data Systems (CCSDS) is an organization officially established by the management of member space Agencies. The Committee meets periodically to address data systems problems that are common to all participants, and to formulate sound technical solutions to these problems. Inasmuch as participation in the CCSDS is completely voluntary, the results of Committee actions are termed RECOMMENDATIONS and are not considered binding on any Agency.

This RECOMMENDATION is issued by, and represents the consensus of, the CCSDS Plenary body. Agency endorsement of this RECOMMENDATION is entirely voluntary. Endorsement, however, indicates the following understandings:

o Whenever an Agency establishes a CCSDS-related STANDARD, this STANDARD will be in accord with the relevant RECOMMENDATION. Establishing such a STANDARD does not preclude other provisions which an Agency may develop.

o Whenever an Agency establishes a CCSDS-related STANDARD, the Agency will provide other CCSDS member Agencies with the following information:

  -- The STANDARD itself.

  -- The anticipated date of initial operational capability.

  -- The anticipated duration of operational service.

o Specific service arrangements shall be made via memoranda of agreement. Neither this RECOMMENDATION nor any ensuing STANDARD is a substitute for a memorandum of agreement.

No later than five years from its date of issuance, this Recommendation will be reviewed by the CCSDS to determine whether it should: (1) remain in effect without change; (2) be changed to reflect the impact of new technologies, new requirements, or new directions; or, (3) be retired or cancelled.

In those instances when a new version of a RECOMMENDATION is issued, existing CCSDS-related Agency standards and implementations are not negated or deemed to be non-CCSDS compatible. It is the responsibility of each Agency to determine when such standards or implementations are to be modified. Each Agency is, however, strongly encouraged to direct planning for its new standards and implementations towards the later version of the Recommendation.

FOREWORD

This document, which is a technical Recommendation prepared by the Consultative Committee for Space Data Systems (CCSDS), is intended for use by participating space Agencies in their development of ‘Advanced Orbiting Systems’.

This Recommendation, written using the ISO Formal Description Technique LOTOS, contains a formal specification of the Path Layer Protocol and Service, described in Natural Language in reference [2]. Annex A contains a set of tests, also written using LOTOS, which specify the required behaviour of the Path Layer Protocol and Service under certain control and input conditions.

The Abstract Data Types used within this document are given in full in reference [4], and the rationale behind the production of this formal specification is given in reference [7].

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Recommendation is therefore subject to CCSDS document management and change control procedures which are defined in reference [1].

At time of publication, the active Member and Observer Agencies of the CCSDS were

Member Agencies

- British National Space Centre (BNSC)/United Kingdom.
- Canadian Space Agency (CSA)/Canada.
- Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
- Centre National d'Etudes Spatiales (CNES)/France.
- Deutsche Forschungsanstalt für Luft- und Raumfahrt e.V. (DLR)/Germany.
- European Space Agency (ESA)/Europe.
- Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
- National Aeronautics and Space Administration (NASA HQ)/USA.
- National Space Development Agency of Japan (NASDA)/Japan.

Observer Agencies

- Australian Space Office (ASO)/Australia.
- Austrian Space Agency (ASA)/Austria.
- Belgian Science Policy Office (SPO)/Belgium.
- Centro Tecnico Aeroespacial (CTA)/Brazil.
- Chinese Academy of Space Technology (CAST)/China.
- Communications Research Laboratory (CRL)/Japan.
- Danish Space Research Institute (DSRI)/Denmark.
- European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
- European Telecommunications Satellite Organization (EUTELSAT)/Europe.
- Hellenic National Space Committee (HNSC)/Greece.
- Indian Space Research Organization (ISRO)/India.
- Industry Canada/Communications Research Center (CRC)/Canada.
- Institute of Space and Astronautical Science (ISAS)/Japan.
- Institute of Space Research (IKI)/Russian Federation.
- KFKI Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
- MIKOMTEK: CSIR (CSIR)/Republic of South Africa.
- Ministry of Communications (MOC)/Israel.
- National Oceanic & Atmospheric Administration (NOAA)/USA.
- Swedish Space Corporation (SSC)/Sweden.
- United States Geological Survey (USGS)/USA.

DOCUMENT CONTROL

Document	Title	Date	Status
CCSDS 705.2-B-1	Recommendation for Space Data Systems Standards—Advanced Orbiting Systems, Networks and Data Links: Formal Specification of the Path Service and Protocol—Addendum to CCSDS 701.0-B-2, Issue 1	May 1994	Original Issue

CONTENTS

REFERENCES

[1] *Procedures Manual for the Consultative Committee for Space Data Systems*. CCSDS A00.0-Y-6. Yellow Book. Issue 6. Washington, D.C.: CCSDS, May 1994 or later issue.

[2] *Advanced Orbiting Systems, Networks and Data Links: Architectural Specification*. Recommendation for Space Data Systems Standards, CCSDS 701.0-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, November 1992 or later issue.

[3] *Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO 8807. Issue 1. Geneva: ISO, 1989.

[4] *Advanced Orbiting Systems, Networks and Data Links: Abstract Data Type Library—Addendum to CCSDS 701.0-B-2*. Recommendation for Space Data Systems Standards, CCSDS 705.1-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, May 1994 or later issue.

[5] *Advanced Orbiting Systems, Networks and Data Links: Formal Specification of the VCLC Service and Protocol—Addendum to CCSDS 701.0-B-2*. Recommendation for Space Data Systems Standards, CCSDS 705.3-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, May 1994 or later issue.

[6] *Advanced Orbiting Systems, Networks and Data Links: Formal Specification of the VCA Service and Protocol—Addendum to CCSDS 701.0-B-2*. Recommendation for Space Data Systems Standards, CCSDS 705.4-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, May 1994 or later issue.

[7] *Advanced Orbiting Systems, Networks and Data Links: Formal Definition of CPN Protocols, Methodology and Approach*. Report Concerning Space Data Systems Standards, CCSDS 705.0-G-2. Green Book. Issue 2. Washington, D.C.: CCSDS, October 1993 or later issue.

[8] *Advanced Orbiting Systems, Networks and Data Links: Summary of Concept, Rationale and Performance*. Report Concerning Space Data Systems Standards, CCSDS 700.0-G-3. Green Book. Issue 3. Washington, D.C.: CCSDS, November 1992 or later issue.

1 INTRODUCTION

1.1 PURPOSE AND SCOPE

This document provides formal specifications of the Consultative Committee for Space Data Systems (CCSDS) Advanced Orbiting Systems (AOS) Path service and protocol[1] using the ISO LOTOS formal description technique (refer to reference [3]). These formal specifications are not intended as replacements for the natural-language specifications provided in the AOS Blue Book (reference [2]), but as unambiguous expressions of those specifications, which may be used to clarify any problem areas.

This document is one of four CCSDS Recommendations that provide LOTOS specifications for the suite of AOS services and protocols (see references [4] through [6]). The relationship between the main AOS Recommendation and the four LOTOS Specifications is shown below; the numbers to the right are the CCSDS document references for the Recommendations containing the LOTOS Specifications.

ADT Library	705.1
Path Service	705.2
Path Protocol	705.2
VCLC Service	705.3
VCLC Protocol	705.3
VCA Service	705.4
VCA Protocol	705.4

A supporting CCSDS Report (reference [7]) contains the rationale, methodology, and approach used to prepare the LOTOS specifications.

These documents are expected to be of use primarily to the technical experts responsible for the design, configuration, and testing of AOS implementations; a basic knowledge of LOTOS is required to understand the formal specifications. Other users of the AOS services should consult the main AOS Recommendation and the companion CCSDS Report (references [2] and [8]).

1.2 BIT ORDERING

Throughout this specification the bit ordering method shown in Figure 1-1 is used; the Most Significant Bit is labelled ‘bit0’ and all subsequent bits are labelled in ascending order.

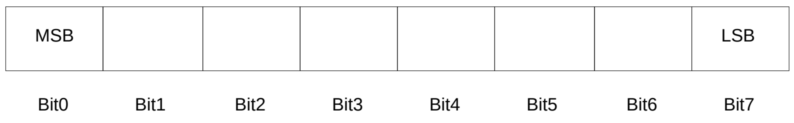

Figure 1-1: Bit Ordering

[1]The natural-language specifications for the Path service and protocol are contained in the AOS Blue Book, CCSDS 701.0-B-2, reference [2].

2 PATH PROTOCOL

```
specification PathProtocol [pkt, oct, man, snw]
                           (packetType : PacketType,
                            dataLossFlag : Bool) : noexit
```

2.1 INTRODUCTION

The CCSDS Path service provides an optimised connectionless data transfer service between a single source user application and one or more destination applications.

2.1.1 Path Architecture

The architecture of the CCSDS Path service with respect to other elements of the CCSDS architecture is given in Figure 2-1. The internal architecture of the Path service used in this specification is given in Figure 2-3.

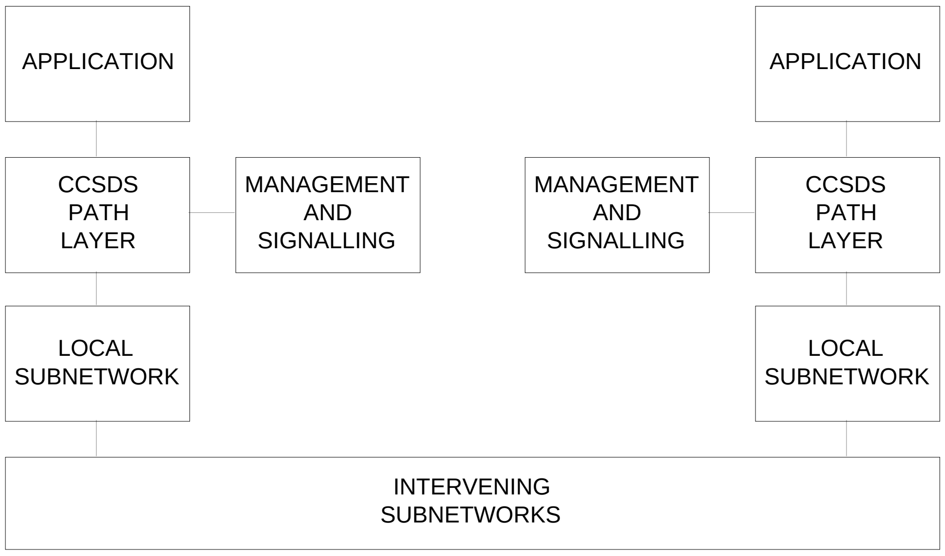

Figure 2-1: Path Service Architecture

Note that this diagram is a very simplistic representation of the Path service architecture, it does, however, provide a useful overview of the CCSDS system. Applications communicate over Logical Data Paths (see section 1.2) provided by the Path service, each application may use many Logical Data Paths, but each Path may be used by only one source application.

2.1.2 Logical Data Paths

The Logical Data Path is the backbone of the path service, it is a single start point, single/multiple endpoint route which transports CCSDS packets unreliably between applications.

Logical Data Paths are referenced by Path IDs; each Path ID is constructed from an Application ID and an optional Application ID Qualifier. Application IDs are 11-bit identifiers which are passed to the Path service by applications wishing to transfer data. Because of the international nature of the CCSDS Recommendations, it was felt that 2048 Logical Data Paths would be insufficient, and therefore the concept of an Application ID (APID) Qualifier was introduced; Path naming domains (typically a spacecraft) can therefore be defined which have access to all 2048 Application IDs, and the Path IDs are formed and kept unique by the addition of the APID Qualifier. The qualifier is represented by addressing mechanisms local to each subnet (the spacecraft ID in the SLS). The APID Qualifier is an optional part of the Path Identifier; if the Qualifier is not to be used, then the value “NullAPIDQual” should be applied.

As is shown in Figure 2-2, Path IDs have a Service Access Point (SAP) like quality; each Path may be used by only one application, but each application may use many Paths (by passing a different Application Process ID in each case), thus applications may have a range of end applications (or sets of end applications) to communicate with.

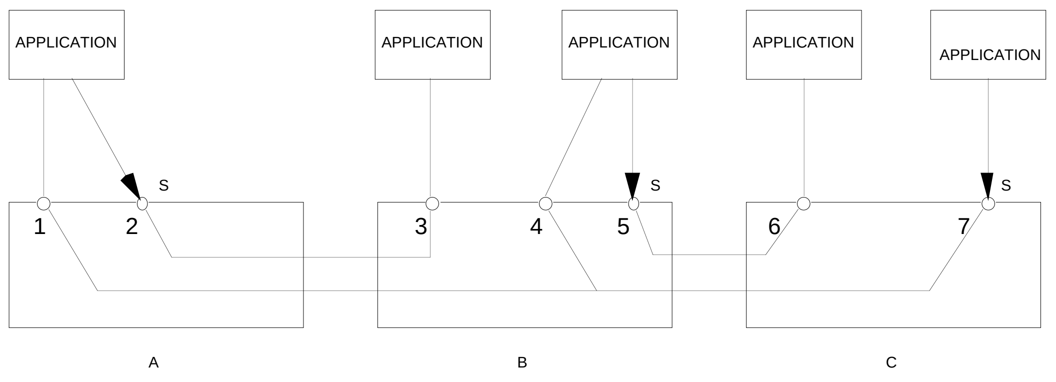

Figure 2-2: Path ID Characteristics

There are several points worth noting in the above diagram:

Three Paths are shown; the start points are labelled with a lower case ‘s’. The first Path runs from the SAP labelled 2 on node A to the SAP labelled 3 on node B. The second Path runs from the SAP labelled 5 on node B to the SAP labelled 6 on node C. The final Path runs from SAP 7 on node C to SAP 1 on node A, and SAP 4 on node B.

The SAP numbers are a convenience; they bear no resemblance to ‘real’ SAP numbers which would effectively be derived from the PathID.

Path entity A does not ‘know’ where the first Path ends, only that Path entity B is the next ‘step’ in the Path; the Path is effectively re-evaluated at each step, B’s ‘idea’ of the first Path is completely different to A’s.

The APID remains constant through the entire length of the path. Packets generated by or destined for different naming domains are kept unique by use of the APID qualifier.

Each Path may end at multiple nodes, but may have only one endpoint at any node; this is a consequence of the SAP nature of the PathIDs.

2.1.3 Specification Architecture

The LOTOS specification is broken down internally as shown in Figure 2-3, note that conformance with a LOTOS specification is measured observationally, that is, when viewed as 'black boxes' any implementation and this specification should respond in the same manner to any stimuli. It is not a requirement that any implementation should explicitly follow the internal breakdown used here.

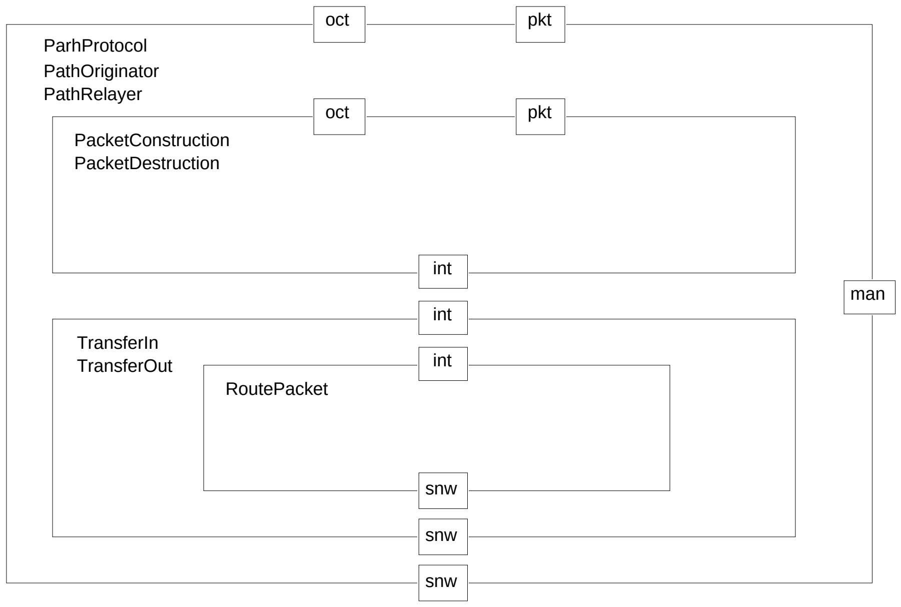

Figure 2-3: LOTOS Specification Internal Breakdown

2.1.4 Management Interface

Two classes of Management Interaction are recognised by the Path Protocol:

Start Path Originator.
Start Path Relayer.

The parameters for these interactions are given below:

The Start Path Originator Interaction has the following parameters:

pathID (APID + optional APIDQualifier)
pathServiceType (Either UserFormatted or OctetString)
initSequenceCount
maxSDULength
localTerminationFlag
terminatingPathServiceType
onwardRelayingTable (See below)

Note that the Maximum SDU Length is configurable by Path - a result of the fact that a PathID may traverse several different subnetworks, and hence is subject to the maximum SDU Length restrictions of all those subnetworks, rather than just the length restrictions of the subnetwork(s) to which the node is immediately attached.

The Start Path Relayer Interaction has the following parameters:

pathID (APID + optional APIDQualifier)
maxSDULength
subnetID
SNSAP
localTerminationFlag
terminatingPathServiceType
onwardRelayingTable (See below)

Note that this message contains an implicit mapping between SNSAPs and APIDQualifiers, thus providing the mechanism for retaining the APIDQualifier over the entire length of the LDP.

The onwardRelayingTable referred to above consists of:

Zero or more of:
- SubnetID
- Source Subnetwork SAP
- Destination Subnetwork SAP

Note - The [man] gate does not represent a true management interface; functions such as stopping, restarting and interrogating the protocol entities have not been specified. The [man] gate interaction only performs instantiation of protocol entities.

2.2 ABSTRACT DATA TYPES

```
library
   PathID, ApidQualifier, CCSDSPacket, Boolean, Bit,
   NaturalNumber, SecondaryHeaderIndicator,
   OctetString, DataLossIndicator
endlib
```

2.2.1 Service Type

The Service type describes the two flavours of service available at the Path SAPs; Packet, where the user passes CCSDS Packets across the service interface and OctetString, where the user passes a block of data across the service interface. The Packet service is referred to in the LOTOS as ‘UserFormatted’ to avoid confusion with other uses of the term ‘Packet’.

```
type    Service is Boolean
sorts   Service
opns    UserFormatted           : -> Service
        OctetString             : -> Service
        _Ne_                    : Service, Service -> Bool
        _Eq_                    : Service, Service -> Bool
eqns    forall S1, S2 : Service

        ofsort Bool

        UserFormatted Eq UserFormatted    = True;

        UserFormatted Eq OctetString      = False;

        OctetString Eq UserFormatted      = False;

        OctetString Eq OctetString        = True;

        S1 Ne S2                          = Not (S1 Eq S2);
endtype
```

2.2.2 Relaying Table

This ADT is part of the management information base for the Path protocol. At the moment this is not the subject of CCSDS standardisation. The following description therefore represents a minimum set of data required to allow the Path protocol to function and should not constrain future implementations.

```
type    RelayingTable is Relay
sorts   RelayingTable
opns    CreateRT                 : -> RelayingTable
        Add                      : Relay, RelayingTable -> RelayingTable
        Tail                     : RelayingTable -> RelayingTable
        Head                     : RelayingTable -> Relay
eqns    forall R1, R2 : Relay, RT : RelayingTable

        ofsort Relay

        Head(CreateRT)                    = NULLRelay;
        Head(Add(R1, CreateRT))           = R1;
        Head(Add(R1, Add(R2, RT)))        = Head(Add(R2, RT));

        ofsort RelayingTable

        Tail(CreateRT)                    = CreateRT;
        Tail(Add(R1, CreateRT))           = CreateRT;
        Tail(Add(R1, Add(R2, RT)))        = Add(R1, Tail(Add(R2, RT)));
endtype
```

2.2.3 Relay Information

This ADT is part of the management information base for the Path protocol. At the moment this is not the subject of CCSDS standardisation. The following description therefore represents a minimum set of data required to allow the Path protocol to function and should not constrain future implementations.

```
type    Relay is SubnetID, SNSAP, Boolean
sorts   Relay
opns    MakeRelay             : SubnetID, SNSAP, SNSAP -> Relay
        NULLRelay             : -> Relay
        GetSubnetID           : Relay -> SubnetID
        GetSourceAddress      : Relay -> SNSAP
        GetDestAddress        : Relay -> SNSAP
        _Eq_, _Ne_            : Relay, Relay -> Bool
eqns    forall SSNSAP, DSNSAP : SNSAP, SID : SubnetID, R1, R2 : Relay

        ofsort SubnetID

        GetSubnetID(MakeRelay(SID, SSNSAP, DSNSAP))      = SID ;

        ofsort SNSAP

        GetSourceAddress(MakeRelay(SID, SSNSAP, DSNSAP)) = SSNSAP ;

        GetDestAddress(MakeRelay(SID, SSNSAP, DSNSAP))   = DSNSAP ;

        ofsort Bool

        NULLRelay Eq MakeRelay(SID, SSNSAP, DSNSAP)      = False ;

        MakeRelay(SID, SSNSAP, DSNSAP) Eq NULLRelay      = False ;

        NULLRelay Eq NULLRelay                           = True ;

           (R1 Ne NULLRelay) And (R2 Ne NULLRelay) =>
        R1 Eq R2        = (GetSubnetID(R1) Eq GetSubnetID(R2)) And
                          (GetSourceAddress(R1) Eq GetSourceAddress(R2)) And
                          (GetDestAddress(R1) Eq GetDestAddress(R2)) ;

        R1 Ne R2        = Not(R1 Eq R2) ;

endtype
```

2.2.4 Subnetwork Identifier

The Subnetwork Identifier is used by Path Entities which are connected to two or more subnetworks; the range of identifiers is limited to 255 in the following specification. However, it is not meant to constrain an implementation to either have a Subnet ID or to use one that is eight bits long.

```
type    SubnetID is Bit, Boolean
sorts   SubnetID
opns    SubnetID        : Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit -> SubnetID
        Bit1, Bit2,
        Bit3, Bit4,
        Bit5, Bit6,
        Bit7, Bit8      : SubnetID -> Bit
        _Eq_, _Ne_      : SubnetID, SubnetID -> Bool
eqns    forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit, SID1, SID2 : SubnetID

        ofsort Bit

        Bit1(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b1;
        Bit2(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b2;
        Bit3(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b3;
        Bit4(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b4;
        Bit5(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b5;
        Bit6(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b6;
        Bit7(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b7;
        Bit8(SubnetID(b1, b2, b3, b4, b5, b6, b7, b8))        = b8;

        ofsort Bool

        SID1 Eq SID2    = (Bit1(SID1) Eq Bit1(SID2)) And
                          (Bit2(SID1) Eq Bit2(SID2)) And
                          (Bit3(SID1) Eq Bit3(SID2)) And
                          (Bit4(SID1) Eq Bit4(SID2)) And
                          (Bit5(SID1) Eq Bit5(SID2)) And
                          (Bit6(SID1) Eq Bit6(SID2)) And
                          (Bit7(SID1) Eq Bit7(SID2)) And
                          (Bit8(SID1) Eq Bit8(SID2));

        SID1 Ne SID2    = Not(SID1 Eq SID2);
endtype
```

```
type    SNSAP is Bit, Boolean
sorts   SNSAP
opns    NullSNSAP       : -> SNSAP
        Add             : Bit, SNSAP -> SNSAP
        _Eq_, _Ne_      : SNSAP, SNSAP -> Bool

eqns    forall b1, b2 : Bit, SNSAP1, SNSAP2 : SNSAP

        ofsort Bool

        NullSNSAP Eq NullSNSAP = True ;

        NullSNSAP Eq Add(b1, SNSAP1) = False ;

        Add(b1, SNSAP1) Eq NullSNSAP = False ;

        Add(b1, SNSAP1) Eq Add(b2, SNSAP2) =
           (b1 Eq b2) And (SNSAP1 Eq SNSAP2) ;

        SNSAP1 Ne SNSAP2 = Not(SNSAP1 Eq SNSAP2) ;

endtype
```

2.3 THE BEHAVIOUR

In the following description, a value of all zeroes for the initial PacketSequence Count is chosen for convenience only. An implementation may choose to initialise this count with some other value.

```
behaviour

PathProtocol [pkt,oct,man,snw] (packetType, dataLossFlag)

where

process PathProtocol [pkt,oct,man,snw] (packetType : PacketType,
                                        dataLossFlag : Bool) : noexit :=
(
    (
        man ? pathID : PathId
            ? pathServiceType : Service
            ? maxSDULength : Nat
            ? onwardRelayingTable : RelayingTable ;
        (
             PathOriginator [pkt, oct, snw]
                           (pathServiceType,
                            pathID,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0),
                            packetType,
                            maxSDULength,
                            onwardRelayingTable)
            |||
            PathProtocol [pkt,oct,man,snw] (packetType, dataLossFlag)
        )
    )
    []
    (
        man ? pathID : PathId
            ? maxSDULength : Nat
            ? subnetID : SubnetID
            ? SNSAP : SNSAP
            ? localTerminationFlag : Bool
            ? terminatingPathServiceType : Service
            ? onwardRelayingTable : RelayingTable ;
        (
            PathRelayer [pkt, oct, snw]
                       (terminatingPathServiceType,
                        pathID,
                        localTerminationFlag,
                        onwardRelayingTable,
                        subnetID,
                        SNSAP,
                        maxSDULength,
                        dataLossFlag)
            |||
            PathProtocol [pkt,oct,man,snw] (packetType, dataLossFlag)
        )
    )
)
endproc PathProtocol
```

The Type field of the packet is assumed to be fixed for a particular Path Layer instantiation. This is not intended to constrain its use in any implementation. The PacketDestruction process corresponds to the Packet Extraction function described in the AOS Blue Book.

```
process PathOriginator [pkt, oct, snw]
                       (pathServiceType : Service,
                        pathID : PathID,
                        initSequenceCount : PacketSequenceCount,
                        packetType : PacketType,
                        maxSDULength : Nat,
                        onwardRelayingTable : RelayingTable) : noexit :=
(
hide int in
    (
         PacketConstruction [pkt,oct,int]
                            (pathServiceType,
                             pathID,
                             initSequenceCount,
                             packetType,
                             maxSDULength)
         |[int]|
         TransferOut [int,snw]
                     (pathID,
                      onwardRelayingTable)
     )
)
endproc PathOriginator
```

```
process PathRelayer [pkt, oct, snw]
                    (terminatingPathServiceType : Service,
                     pathID : PathID,
                     localTerminationFlag : Bool,
                     onwardRelayingTable : RelayingTable,
                     subnetID : SubnetID,
                     SNSAP : SNSAP,
                     maxSDULength : Nat,
                     dataLossFlag : Bool) : noexit :=
(
hide int in
    (
          PacketDestruction [pkt,oct,int]
                            (terminatingPathServiceType,
                             pathID,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0),
                             True,
                             dataLossFlag)
          |[int]|
          TransferIn [int,snw]
                     (pathID,
                      localTerminationFlag,
                      onwardRelayingTable,
                      subnetID,
                      SNSAP,
                      QualifierPart(pathID),
                      maxSDULength)
    )
)
endproc PathRelayer
```

2.3.1 PacketConstruction Process

This process accepts primitives on the [pkt] and [oct] gate, constructing packets in the case of the OctetString service, or checking their format in the case of the UserFormatted service. These packets are passed through the [int] gate along with the PathID to be relayed onwards as necessary.

The process is passed five parameters:

PathServiceType	determines which gate the process will accept primitives on.
PathID	the PathID (i.e., SAP) to accept primitives on.
SequenceCount	used only by the OctetString service, this is updated with each Packet sent, the initial value being set by management.
packetType	the value to be placed into the Packet Type field of Packets constructed on the OctetString service.
maxSDULength	the maximum length for SDUs on this Path; taken to be length of OctetString for the OctetString service, or total length of Packet for UserFormatted service.

The variable called ‘Data’ at the [oct] gate refers to the O_SDU which is part of the OCTET_STRING.request primitive.

```
process PacketConstruction [pkt,oct,int]
                           (pathServiceType : Service,
                            pathID : PathID,
                            SequenceCount : PacketSequenceCount,
                            packetType : PacketType,
                            maxSDULength : Nat) : noexit :=
(
     let SAP : PathId = pathID in
     (
     [pathServiceType Eq OctetString] ->
          (
          oct ! SAP
              ? Path : PathID
              ? SHI  : SecondaryHeaderIndicator
              ? Data : OctetString [LengthOf(Data) Le maxSDULength] ;

          int ! Path
              ! MakeCCSDSPacket(
                   MakePrimaryHeader(
                      MakePacketID(version1, packetType,
                                    SHToSHF(SHI), APIDPart(Path)),
                      MakePacketSC(PacketSequenceUnSeg, SequenceCount),
                      ConvertNatToPL(Pred(LengthOf(Data)))),
                   Data) ;
          PacketConstruction [pkt,oct,int]
                             (pathServiceType,
                              pathID,
                              Next(SequenceCount),
                              packetType,
                              maxSDULength)
          )
     []
```

NOTE – The APID Qualifier is an optional part of the Path Identifier; if the Qualifier is not to be used, then the value “NullAPIDQual” should be applied.

```
     [pathServiceType Eq UserFormatted] ->
          (
          pkt ! SAP
              ? Qual : APIDQual
              ? Packet : CCSDSPacket
          [(GetPacketLength(GetPrimaryHeader(Packet)) eq
            ConvertNatToPL(Pred(LengthOf(GetUserData(Packet)))))
            and ((LengthOf(GetUserData(Packet))+6) Le maxSDULength)
            and (GetVersion(GetPrimaryHeader(Packet)) eq Version1)
            and (UserAPID(GetAPID(GetPrimaryHeader(Packet))))] ;

          int ! PathID ! Packet;
          PacketConstruction [pkt,oct,int]
                             (pathServiceType,
                              pathID,
                              Next(SequenceCount),
                              packetType,
                              maxSDULength)
          )
     )
)
endproc PacketConstruction
```

2.3.2 PacketDestruction Process

This process accepts packets from the RoutePacket process on the [int] gate. It generates indications according to the Service parameter, checking for violations of the SequenceCount in the OctetString service and generating Data Loss Flags as required.

The process is passed five parameters:

terminatingPathServiceType	determines which gate the process will generate primitives on.
Path	the PathID (i.e., SAP) to generate primitives on.
SequenceCount	only used for the OctetString service; this gives the value of the next expected sequence count and is used in the generation of Data Loss Flags.
First	only used for the OctetString service; if this is set to True, then the process will accept the sequence count in the next packet regardless of the value of SequenceCount.
dataLossFlag	only used for the OctetString service; if this is set to True, then the process will generate Data Loss Flags.

The PacketDestruction process corresponds to the Packet Extraction function described in the AOS Blue Book.

```
process PacketDestruction [pkt,oct,int]
                          (terminatingPathServiceType : Service,
                           Path                       : Pathid,
                           SequenceCount              : PacketSequenceCount,
                           First                      : Bool,
                           dataLossFlag               : Bool) : noexit :=
let SAP : PathId = Path in
(
   int ! Path ? Packet : CCSDSPacket;
   (
      [terminatingPathServiceType Eq UserFormatted] ->
      (
         pkt ! SAP
             ! QualifierPart(Path)
             ! Packet;
         PacketDestruction [pkt,oct,int]
                           (terminatingPathServiceType,
                            Path, SequenceCount,
                            First, dataLossFlag)
      )
      []
      [terminatingPathServiceType Eq OctetString] ->
      (
            [(GetPacketSequenceCount(GetPrimaryHeader(Packet)) Ne SequenceCount)
             and Not(First) and (dataLossFlag Eq True)] ->
            (
            oct ! SAP
                ! Path
                ! SHFToSH(GetSHF(GetPrimaryHeader(Packet)))
                ! GetUserData(Packet)
                ! OSDULost ;
            PacketDestruction [pkt,oct,int]
                        (terminatingPathServiceType, Path,
                         Next(GetPacketSequenceCount(GetPrimaryHeader(Packet))),
                         False, dataLossFlag)
            )
            []
            [((GetPacketSequenceCount(GetPrimaryHeader(Packet))
             Eq SequenceCount) Or (First)) And (dataLossFlag Eq True)] ->
            (
            oct ! SAP
                ! Path
                ! SHFToSH(GetSHF(GetPrimaryHeader(Packet)))
                ! GetUserData(Packet)
                ! OSDUNotLost ;
            PacketDestruction [pkt,oct,int]
                        (terminatingPathServiceType, Path,
                         Next(GetPacketSequenceCount(GetPrimaryHeader(Packet))),
                         False, dataLossFlag)
            )
            []
            [dataLossFlag Eq False] ->
            (
            oct ! SAP
                ! Path
                ! SHFToSH(GetSHF(GetPrimaryHeader(Packet)))
                ! GetUserData(Packet) ;
            PacketDestruction [pkt,oct,int]
                              (terminatingPathServiceType, Path,
                               SequenceCount, False, dataLossFlag)
            )
      )
   )
)
endproc PacketDestruction
```

2.3.3 TransferIn Process

This process accepts Subnetwork Indication primitives from a particular network on a particular SAP, checking the format of the received SDU (i.e., the CCSDS Version 1 packet) and then behaving as the RoutePacket process before re-enabling itself.

The process is passed seven parameters:

Path	the Path on which the process is working.
localTerminationFlag	a Boolean flag indicating whether the Path has a termination at this entity.
onwardRelayingTable	a list of SSNSAP, DSNSAP and SubnetID tuples.
Subnet	the Subnetwork to accept primitives on.
Dest	the Subnetwork SAP at which to accept primitives.
Qual	the APIDQualifier part of the Path on which the process is working.
maxSDULength	the maximum length for Path PDUs on this Path; taken to be the total length of the Packet contained in the subnetwork SDU.

```
process TransferIn [int, snw]
                   (Path : PathId,
                    localTerminationFlag : Bool,
                    onwardRelayingTable : RelayingTable,
                    Subnet : SubnetID,
                    Dest : SNSAP,
                    Qual : APIDQual,
                    MaxSDULength : Nat) : noexit :=
(
        snw ! Subnet
            ? Source : SNSAP
            ! Dest
            ? Packet : CCSDSPacket
            [(MakePathID(GetAPID(GetPrimaryHeader(Packet)),Qual) Eq Path)
         and (GetPacketLength(GetPrimaryHeader(Packet)) Eq
              ConvertNatToPL(Pred(LengthOf(GetUserData(Packet)))))
         and ((LengthOf(GetUserData(Packet))+6) Le MaxSDULength)
         and (GetVersion(GetPrimaryHeader(Packet)) Eq Version1)
         and (UserAPID(GetAPID(GetPrimaryHeader(Packet))))] ;

        RoutePacket [int, snw] (localTerminationFlag,
                                onwardRelayingTable,
                                Packet,
                                Path)
        >> TransferIn [int, snw] (Path,
                                  localTerminationFlag,
                                  onwardRelayingTable,
                                  Subnet,
                                  Dest,
                                  Qual,
                                  MaxSDULength)
)
endproc TransferIn
```

2.3.4 TransferOut Process

This process accepts packets on the [int] gate, and then behaves as the RoutePacket process before re-enabling itself.

The process is passed two parameters:

Path	the Path on which the process is working.
onwardRelayingTable	a list of SSNSAP, DSNSAP and SubnetID tuples.

```
process TransferOut [int, snw]
                    (Path : PathId,
                     onwardRelayingTable : RelayingTable) : noexit :=

(
        int ! Path ? Packet : CCSDSPacket;
        RoutePacket [int, snw] (False,
                                onwardRelayingTable,
                                Packet,
                                Path)
        >> TransferOut [int, snw] (Path, onwardRelayingTable)
)
endproc
```

2.3.5 RoutePacket Process

This process takes relaying information and accordingly generates subnetwork requests, or places the packet on the [int] gate, exiting when the relaying information has been exhausted.

The process is passed four parameters:

LocalTermination	a Boolean type which determines whether the packet should be placed on the [int] gate to be passed out to an application by another process.
onwardRelayingTable	a list of SSNSAP, DSNSAP and SubnetID tuples.
Packet	the packet to be relayed.
Path	the path on which the process is operating.

```
process RoutePacket [int, snw] (LocalTermination : Bool,
                                onwardRelayingTable : RelayingTable,
                                Packet : CCSDSPacket,
                                Path : PathID) : exit :=
(
   [LocalTermination] ->
   (
      int ! Path ! Packet ;
      RoutePacket[int, snw] (False, onwardRelayingTable, Packet, Path)
   )
[]
   [Not(LocalTermination)] ->
   (
      [Head(onwardRelayingTable) Ne NULLRelay] ->
      (
         snw ! GetSubnetID(Head(onwardRelayingTable))
             ! GetSourceAddress(Head(onwardRelayingTable))
             ! GetDestAddress(Head(onwardRelayingTable))
             ! Packet;
         RoutePacket [int, snw] (false,
                                 Tail(onwardRelayingTable),
                                 Packet,
                                 Path)
      )
      []
      [Head(onwardRelayingTable) Eq NULLRelay] -> exit
   )
)
endproc RoutePacket

endspec
```

3 PATH SERVICE

```
specification PathService [pkt, oct, man] : noexit
```

3.1 INTRODUCTION

The LOTOS specification is broken down internally as shown in Figure 3-1, note that conformance with a LOTOS specification is measured observationally, that is, when viewed as 'black boxes' any implementation and this specification should respond in the same manner to any stimuli. It is not a requirement that any implementation should explicitly follow the internal breakdown used here.

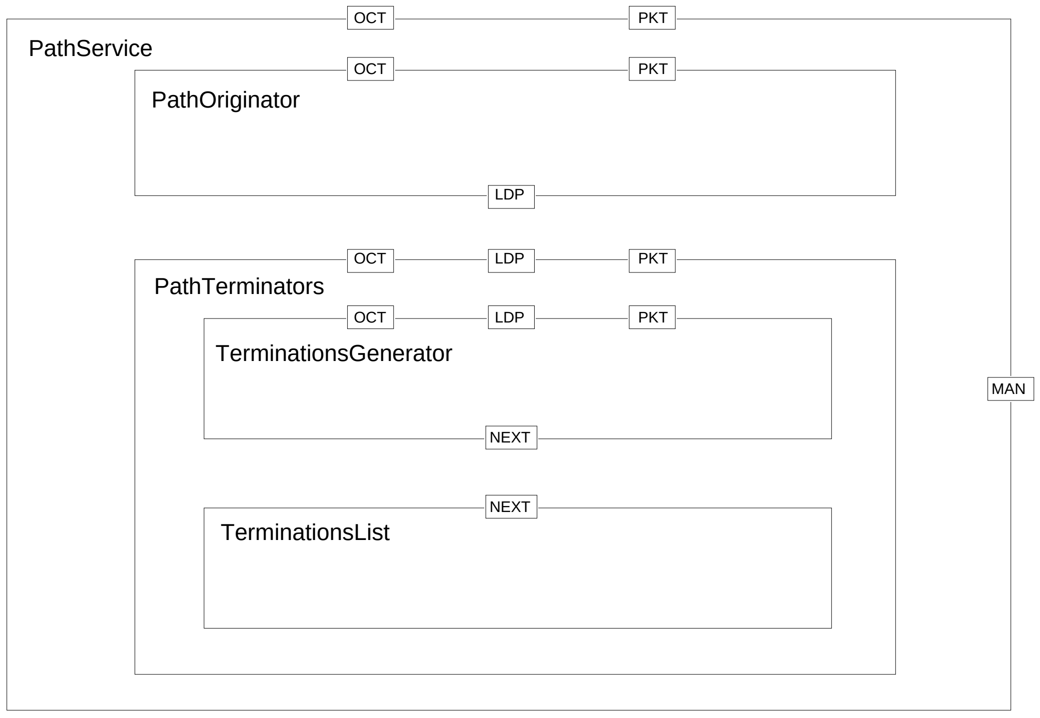

Figure 3-1: LOTOS Specification Internal Breakdown

3.2 ABSTRACT DATA TYPES

```
library CCSDSPacket, PathID, Boolean, NaturalNumber,
        Bit, SecondaryHeaderIndicator, DataLossIndicator endlib
```

3.2.1 Service Type

The Service type describes the two flavours of service available at the Path SAPs; Packet, where the user passes CCSDS Packets across the service interface and OctetString, where the user passes a block of data across the service interface. The Packet service is referred to in the LOTOS as ‘UserFormatted’ to avoid confusion with other uses of the term ‘Packet’.

```
type    Service is Boolean
sorts   Service
opns    UserFormatted           : -> Service
        OctetString             : -> Service
        _Ne_                    : Service, Service -> Bool
        _Eq_                    : Service, Service -> Bool
eqns    forall S1, S2 : Service

        ofsort Bool

        UserFormatted Eq UserFormatted     = True;

        UserFormatted Eq OctetString       = False;

        OctetString Eq UserFormatted       = False;

        OctetString Eq OctetString         = True;

        S1 Ne S2                           = Not (S1 Eq S2);

endtype
```

3.2.2 Packet Queue

```
type  PacketQueue is CCSDSPacket, Boolean
sorts PacketQueue
opns  Create           : -> PacketQueue
      AddBack          : CCSDSPacket, PacketQueue -> PacketQueue
      First            : PacketQueue -> CCSDSPacket
      RemoveFirst      : PacketQueue -> PacketQueue
      _Eq_, _Ne_       : PacketQueue, PacketQueue -> Bool
eqns  forall P1 : CCSDSPacket, PQ1, PQ2 : PacketQueue

      ofsort PacketQueue

      RemoveFirst(Create)               = Create ;
      RemoveFirst(AddBack(P1, Create)) = Create ;
      RemoveFirst(AddBack(P1, PQ1))    = AddBack(P1, RemoveFirst(PQ1)) ;

      ofsort CCSDSPacket

      First(AddBack(P1, Create))       = P1 ;
      First(AddBack(P1, PQ1))          = First(PQ1) ;

      ofsort Bool

      Create Eq Create                  = True ;
      Create Eq AddBack(P1, Create)    = False ;
      Create Eq AddBack(P1, PQ1)       = False ;
      AddBack(P1, Create) Eq Create    = False ;
      AddBack(P1, PQ1) Eq Create       = False ;

      PQ1 Ne PQ2                        = Not(PQ1 Eq PQ2) ;

endtype
```

3.2.3 Endpoint definition

```
type  Endpoint is Service, Boolean
sorts endpoint
opns  MakeEndpoint : Service, Bool -> Endpoint
      Service      : Endpoint -> Service
      DataLossFlag : Endpoint -> Bool
eqns  forall s : Service, b : Bool

   ofsort Service

   Service(MakeEndpoint(s,b))      = s ;

   ofsort Bool

   DataLossFlag(MakeEndpoint(s,b)) = b ;

endtype
```

3.2.4 Path Terminations List

```
type Terminations is Endpoint, Boolean

sorts Terminations

opns
   CreateTL : -> Terminations
   Add      : Endpoint, Terminations -> Terminations
   Head     : Terminations -> Endpoint
   Tail     : Terminations -> Terminations
   _Eq_, _Ne_ : Terminations, Terminations -> Bool

eqns
   forall e : Endpoint, t : Terminations

   ofsort Endpoint

   Head(Add(e, t)) = e ;

   ofsort Terminations

   Tail(Add(e, t)) = t ;

   ofsort Bool

   CreateTL Eq Add(e,t) = False ;
   Add(e,t) Eq CreateTL = False ;
   CreateTL Eq CreateTL = true ;

   CreateTL Ne Add(e,t) = True ;
   Add(e,t) Ne CreateTL = True ;
   CreateTL Ne CreateTL = False ;

endtype
```

3.3 THE BEHAVIOUR

In the following description, a value of all zeroes for the initial PacketSequence Count is chosen for convenience only. An implementation may chose to initialise this count with some other value. The Type field of the packet is assumed to be fixed for a particular Path Layer instantiation. This is not intended to constrain its use in any implementation.

```
behaviour
        PathService [pkt,oct,man]

where

process PathService [pkt,oct,man] : noexit :=

(
     man ? pathID : PathId
         ? sourceService : Service
         ? terminations : Terminations
         ? packetType : PacketType
         ? maxSDULength : Nat ;
     (
          PathService [pkt,oct,man]
          |||
          (
               hide ldp in
               (
                    PathOriginator [pkt,oct,ldp]
                                   (sourceService,
                                    pathID,
                                    PacketSequenceCount(0,0,0,0,0,0,0,
                                                        0,0,0,0,0,0,0),
                                    packetType,
                                    maxSDULength)
                    |[ldp]|
                    PathTerminators [ldp, oct, pkt] (terminations,
                                                     pathID)

               )
          )
     )
)
endproc PathService
```

3.3.1 PathOriginator Process

This process accepts primitives on the [pkt] and [oct] gates, constructing packets in the case of the OctetString service, or checking their format in the case of the UserFormatted service.

The process is passed five parameters:

sourceService	the service offered at the origination.
pathID	the PathID (SAP) on which to accept primitives.
initSequenceCount	used only by the OctetString service, this is updated with each Packet sent.
packetType	the value to be placed into the Packet Type field of Packets constructed on the OctetString service.
maxSDULength	the maximum length for SDUs on this Path; taken to be length of OctetString for the OctetString service, or total length of Packet for UserFormatted service.

The variable called 'Data' at the [oct] gate refers to the O_SDU which is part of the OCTET_STRING.request primitive.

```
process PathOriginator [pkt,oct,ldp] (sourceService : Service,
                                      pathID : PathID,
                                      sequenceCount : PacketSequenceCount,
                                      packetType : PacketType,
                                      maxSDULength : Nat) : noexit :=
(
   let SAP : PathId = PathID in
   (
      [sourceService Eq OctetString] ->
      (
         oct ! SAP
             ? path : PathID
             ? SH : SecondaryHeaderIndicator
             ? Data : OctetString [LengthOf(Data) Le maxSDULength];

         ldp ! MakeCCSDSPacket
                  (MakePrimaryHeader
                      (MakePacketID(Version1, packetType,
                                      SHToSHF(SH), APIDPart(path)),
                       MakePacketSC(PacketSequenceUnSeg, sequenceCount),
                       ConvertNatToPL(Pred(LengthOf(Data)))),
                   Data) ;
         PathOriginator [pkt,oct,ldp] (sourceService,
                                       pathID,
                                       Next(sequenceCount),
                                       packetType,
                                       maxSDULength)
      )
   []
```

```
      [sourceService Eq UserFormatted] ->
      (
         pkt ! SAP
             ? Qual : APIDQual
             ? Packet : CCSDSPacket
         [(GetPacketLength(GetPrimaryHeader(Packet)) eq
           ConvertNatToPL(Pred(LengthOf(GetUserData(Packet)))))
      and ((LengthOf(GetUserData(Packet))+6) Le MaxSDULength)
      and (GetVersion(GetPrimaryHeader(Packet)) eq Version1)
      and (UserAPID(GetAPID(GetPrimaryHeader(Packet))))] ;

         ldp ! Packet ;
         PathOriginator [pkt,oct,ldp] (sourceService,
                                        pathID,
                                        sequenceCount,
                                        packetType,
                                        maxSDULength)
      )
   )
)
endproc PathOriginator
```

3.3.2 PathTerminators Process

The process is passed two parameters:

Path	the PathID (i.e., SAP) on which to generate primitives.
Terminations	a list of Service, Boolean pairs representing the service and data loss flag option for each LDP endpoint.

```
process PathTerminators [ldp,oct,pkt] (Terminations : Terminations,
                                       Path   : Pathid) : noexit :=
(
     hide next in
     (
          TerminationsList [next] (Terminations, Path)
          |[next]|
          TerminationsGenerator [next, pkt, oct, ldp]
     )
)

where

process TerminationsList [next] (Terminations : Terminations,
                                 Path : PathID) : noexit :=
(
[Terminations Eq CreateTL] ->
     stop
[]
[Terminations Ne CreateTL] ->
     (
          next ! Path
               ! Head(Terminations) ;
          TerminationsList [next] (Tail(Terminations), Path)
     )
)
endproc TerminationsList
```

```
process TerminationsGenerator [next, pkt, oct, ldp] : noexit :=
(
     next ? Path : PathID
          ? Termination : Endpoint ;
     (
          hide losepacket in
          (
          PathTerminator [pkt, oct, ldp, losepacket]
                         (Termination,
                          Path,
                          create,
                          OSDUNotLost)
          |[ldp]|
          TerminationsGenerator [next, pkt, oct, ldp]
          )
     )
)


where

process PathTerminator [pkt, oct, ldp, losepacket]
                       (Termination : EndPoint,
                        Path : PathID,
                        Queue : PacketQueue,
                        LossFlag : DataLossIndicator) : noexit :=
(
     (
          ldp ? Packet : CCSDSPacket ;
          PathTerminator [pkt, oct, ldp, losepacket]
                         (Termination,
                          Path,
                          AddBack(Packet, Queue),
                          LossFlag)
     )
     []
     [Queue Ne Create] ->
     (
          [Service(Termination) Eq OctetString] ->
          (
              [DataLossFlag(Termination) Eq False] ->
              (
                   (
                        oct ! Path
                            ! Path
                            ! SHFToSH(GetSHF(GetPrimaryHeader(First(Queue))))
                            ! GetUserData(First(Queue)) ;
                        PathTerminator [pkt, oct, ldp, losepacket]
                                       (Termination,
                                        Path,
                                        RemoveFirst(Queue),
                                        LossFlag)
                   )
                   []
                   (
                        losepacket ;
                        PathTerminator [pkt, oct, ldp, losepacket]
                                       (Termination,
                                        Path,
                                        RemoveFirst(Queue),
                                        LossFlag)
                   )
              )
              []
                          [DataLossFlag(Termination) Eq True] ->
                          (
                               (
                                    oct ! Path
                                        ! Path
                                        ! SHFToSH(GetSHF(GetPrimaryHeader(First(Queue))))
                                        ! GetUserData(First(Queue))
                                        ! LossFlag ;
                                    PathTerminator [pkt, oct, ldp, losepacket]
                                                   (Termination,
                                                    Path,
                                                    RemoveFirst(Queue),
                                                    OSDUNotLost)
                               )
                               []
                               (
                                    losepacket ;
                                    PathTerminator [pkt, oct, ldp, losepacket]
                                                   (Termination,
                                                    Path,
                                                    RemoveFirst(Queue),
                                                    OSDULost)
                               )
                          )
                      )
                      []
                      [Service(Termination) Eq UserFormatted] ->
                      (
                           (
                                pkt ! Path
                                    ! QualifierPart(Path)
                                    ! First(Queue) ;
                                PathTerminator [pkt, oct, ldp, losepacket]
                                               (Termination,
                                                Path,
                                                RemoveFirst(Queue),
                                                LossFlag)
                           )
                           []
                           (
                                losepacket ;
                                PathTerminator [pkt, oct, ldp, losepacket]
                                               (Termination,
                                                Path,
                                                RemoveFirst(Queue),
                                                LossFlag)
                           )
                      )
                 )
            )
            endproc PathTerminator

            endproc TerminationsGenerator

            endproc PathTerminators

            endspec
```

ANNEX A

PATH PROTOCOL AND SERVICE TESTS

(THIS ANNEX **IS NOT** PART OF THE RECOMMENDATION)

Purpose:

This Annex details procedures for testing the Path protocol and service.

<u>Path Protocol Test 1</u>

This test attempts to send an octet string on a path which has not been set up as an octet source.

Run the test using the command:

```
ts pp.t1.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt1 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test.

This is just a dummy process used to contain the test behaviour:

```
process ppt1 (packetType : PacketType,
               dataLossFlag : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (packetType, dataLossFlag)
   )
)

where
```

This is the test behaviour:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(

(
```

Attempt to send an octet string on a path which has not yet been set up. Acceptance of this event leads to failure.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,0),
                   AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;

   failure ; exit
)
[]
(
```

No choice should be offered. Only the success event should be possible as the octet string event above should be rejected.

```
   success ; exit
)
)

endproc
endproc
```

Path Protocol Test 1a

This test attempts to send an CCSDS Packet on a path which has not been set up as an packet source.

Run the test using the command:

```
ts pp.t1a.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt1a from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test.

This is just a dummy process used to contain the test behaviour:

```
process ppt1a (packetType : PacketType,
               dataLossFlag : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (packetType, dataLossFlag)
   )
)

where
```

This is the test behaviour:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(

(
```

Attempt to send an octet string on a path which has not yet been set up. Acceptance of this event leads to failure.

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,0,1,0)),
               MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
            AddFront(Octet(0,0,0,0,0,0,0,0),
                   AddFront(Octet(0,0,0,0,0,0,0,1), NullOS))) ;

   failure ; exit
)
[]
(
```

No choice should be offered. Only the success event should be possible as the octet string event above should be rejected.

```
   success ; exit
)
)

endproc
endproc
```

Path Protocol Test 2

This test attempts to send a user-formatted packet on a path set up as an octet source. This situation should be disallowed by the protocol.

Run the test using the command:

```
ts pp.t2.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt2 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should only be selected when there are no alternatives.

This process is just a holder for the test behaviour:

```
process ppt2 (pType : PacketType,
              dataLossFlag : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlag)
   )
)
```

```
where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Octetstring
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;


((
```

Attempt to send a user-formatted packet on an octet string path. This request should be rejected by the protocol.

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
             AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
 )
[]
(
```

As the packet request should be rejected, this is the only event that should be offered, indicating success of the test.

```
   success ; exit
))
)
endproc
endproc
```

Path Protocol Test3

This test attempts to send a user-formatted packet on a path which is not set up as a source, only as a router.

Run the test using the command:

```
ts pp.t3.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt3 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test. The exception to this is that the success event should not be selected if there are alternatives.

This process is just a holder for the test behaviour:

```
process ppt3 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide failure, success in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! False
       ! UserFormatted
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
```

```
((
```

Attempt to send a user-formatted packet on a path not formatted as a source, just a router. The request should be rejected.

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,1)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
   failure ; exit
 )
[]
 (
```

As the above packet request should be rejected only this success event should be offered.

```
   success ; exit
))
)
endproc
endproc
```

Path Protocol Test 3a

This test attempts to send an octet string on a path which is not set up as a source, only a relayer.

Run the test using the command:

```
ts pp.t3a.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt3a from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test. The exception to this is that the success event should not be selected if there are alternatives.

This process is just a holder for the test behaviour:

```
process ppt3a (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide failure, success in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                   Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! False
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                      Add(1, Add(0, NullSNSAP)),
                      Add(1, NullSNSAP)),
             CreateRT) ;
```

```
((
```

Attempt to send an octet string on a path not formatted as a source, just a router. The request should be rejected.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                    Add(1, NullAPIDQual))
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
   failure ; exit
 )
[]
 (
```

As the above octet string request should be rejected only this success event should be offered.

```
   success ; exit
))
)
endproc
endproc
```

Path Protocol Test4

This test attempts to send user-formatted packets on a path which is set up as a packet source, only the valid packet should be accepted, leaving the badly formatted ones.

It also tests that the secondary header flag, the sequence count, the data length, the data, the packet type and the sequence flag fields are all carried through correctly.

Run the test using the command:

```
ts pp.t4.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt4 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should only be selected when there are no alternatives.

This process holds the test behaviour:

```
process ppt4 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)
where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! UserFormatted
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;

   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! False
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;

   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! UserFormatted
       ! CreateRT ;
```

Attempt to send a badly user-formatted packet on a path set up as a packet source.

```
(
(
```

A packet where the APID is not a user APID:

```
   pkt ! MakePathID(APID(1,1,1,1,1,1,1,1,1,1,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(1,1,1,1,1,1,1,1,1,1,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the version is not version 1

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version(0,0,1),
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the packet length is not consistent with data length:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which is valid

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

results in the following subnetwork indication:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Another valid indication to show that the relevant details are carried through correctly

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                  Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceFirstSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;
```

results in the following subnetwork indication:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceFirstSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;

   success ; exit
))
)
endproc
endproc
```

Path Protocol Test 4a

This test attempts to send an over-sized user-formatted packet on a path which is set up as a packet source.

Run the test using the command:

```
ts pp.t4a.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt4a from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should only be selected when there are no alternatives.

This process holds the test behaviour:

```
process ppt4a (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! UserFormatted
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
           CreateRT) ;

   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! False
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
           CreateRT) ;

   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! UserFormatted
       ! CreateRT ;
```

Attempt to send a badly user-formatted packet on a path set up as a packet source.

```
(
(
```

A packet which exceeds the MaxSDU parameter:

```
   pkt ! MakePathID(APID(1,1,1,1,1,1,1,1,1,1,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
          (MakePrimaryHeader
             (MakePacketID(Version1,
                           PacketType(0),
                           SHF(0),
                           APID(1,1,1,1,1,1,1,1,1,1,1)),
             MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
         AddFront(Octet(0,0,0,0,0,0,0,1),
         AddFront(Octet(0,0,0,0,0,0,1,0),
         AddFront(Octet(0,0,0,0,0,0,1,1),
         AddFront(Octet(0,0,0,0,0,1,0,0),
         AddFront(Octet(0,0,0,0,0,1,0,1), NullOS)))))) ;

   failure ; exit
)
[]
(
   success ; exit
)
)
)

endproc
endproc
```

Path Protocol Test5

This test injects packets into a path entity from the subnetwork and checks that they emerge on the local node.

Run the test using the command:

```
ts pp.t5.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt5 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be selected unless there is no alternative.

This process is just a holder for the test behaviour:

```
process ppt5 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! UserFormatted
       ! CreateRT ;
```

Inject packets into path from the subnetwork on LDP 4,1. Note that packets can come from any source SAP.

```
((
```

This is the valid indication:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which should result in the following packet indication:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

A packet which has come from an unrecognised SN_SAP. This is a valid situation

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, Add(0, Add(1, NullSNSAP))))
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

resulting again in a valid indication:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   success ; exit

)
[]
(
```

A packet containing an APID which is not a user APID. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(1,1,1,1,1,1,1,1,1,1,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet on an unconfigured subnetwork. This should be rejected.

```
   snw ! SubnetID(0,0,0,1,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
```

```
(
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version(0,0,1),
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the packet length is not consistent with data. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which has arrived on an unrecognised SAP. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, Add(0, Add(1, Add(0, NullSNSAP)))))
       ! MakeCCSDSPacket
         (MakePrimaryHeader
            (MakePacketID(Version1,
                           PacketType(0),
                           SHF(0),
                           APID(0,0,0,0,0,0,0,0,1,0,0)),
            MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
            PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)))
endproc
endproc
```

<u>Path Protocol Test 5a</u>

This test injects invalid subnetwork.indications into a path entity from the subnetwork and checks that they are disallowed.

Run the test using the command:

```
ts pp.t5a.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt5a from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be selected unless there is no alternative.

This process is just a holder for the test behaviour:

```
process ppt5a (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! UserFormatted
       ! CreateRT ;
```

Inject packets into path from the subnetwork on LDP 4,1

```
(
(
```

A packet where the APID has not been set up by management. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which exceeds the MaxSDU parameter. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, Add(0, Add(1, Add(0, NullSNSAP)))))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
   success ; exit
)
)
)
endproc
endproc
```

Path Protocol Test6

This test injects packets into path from the subnetwork and checks that they emerge on the local node as octet strings.

Run the test using the command:

```
ts pp.t6.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt6 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is the success event which should not be chosen if there are alternatives.

This is just a holder for the test behaviour:

```
process ppt6 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, False)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! OctetString
       ! CreateRT ;
```

Inject packets into path from the subnetwork on LDP 4,1. Note that packets can come from any source SAP.

```
((
```

This is the valid indication:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which should result in the following indication:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

A packet which has come from an unrecognised SN_SAP. This is a valid situation

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, Add(0, Add(0, Add(1, NullSNSAP)))))
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,1,1),
          AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))) ;
```

which should also result in an indication:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,1,1),
                   AddFront(Octet(0,0,0,0,0,1,0,0), NullOS)) ;

   success ; exit

)
[]
```

```
(
```

A packet containing an APID which is not a user APID:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(1,1,1,1,1,1,1,1,1,1,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet on an unconfigured subnetwork:

```
   snw ! SubnetID(0,0,0,1,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
```

(

A packet where the version is not version 1:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version(0,0,1),
                             PacketType(0),
                             SHAbsent,
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the packet length is not consistent with data:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHAbsent,
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which has arrived on an unrecognised SAP:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, Add(0, Add(1, Add(0, NullSNSAP)))))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHAbsent,
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)))
endproc
endproc
```

Path Protocol Test7

This test injects packets into path from the subnetwork and checks that they emerge on the local node and are re-transmitted. It is a test of the routing function.

Run the test using the command:

```
ts pp.t7.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt7 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is the success event which should not be chosen if an alternative exists.

This is just a holder for the test behaviour:

```
process ppt7 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, False)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
```

Inject packets into path from the subnetwork on LDP 1,2. Note that packets can come from any source SAP.

```
((
```

This is the valid indication:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,0,0,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which results in the following octet service indication:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

and the following subnetwork request:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,0,0,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   success ; exit

)
[]
(
```

A packet containing an APID which is not a user APID, rejected:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(1,1,1,1,1,1,1,1,1,1,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet on an unconfigured subnetwork, rejected.

```
   snw ! SubnetID(0,0,0,1,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the version is not version 1, rejected:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version(0,0,1),
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
```

```
(
```

A packet where the packet length is not consistent with data, rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
               MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which has arrived on an unrecognised SAP, rejected:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, Add(0, Add(1, Add(0, NullSNSAP)))))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
               MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)))
endproc
endproc
```

Path Protocol Test8

This test injects packets into path from the subnetwork and checks that they emerge on the local node and are re-transmitted if necessary.

It demonstrates the path layer's ability to handle multiple paths.

Run the test using the command:

```
ts pp.t8.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt8 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be chosen unless no alternative exists.

This is just a holder for the test behaviour:

```
process ppt8 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, False)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! OctetString
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
```

```
man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,1),
                 Add(1, NullAPIDQual))
    ! Succ(Succ(8))
    ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, Add(0, NullSNSAP))
    ! False
    ! OctetString
    ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullSNSAP)),
                    Add(1, NullSNSAP)),
        CreateRT) ;
```

Inject packets into path from the subnetwork on LDP 1,2. Note that packets can come from any source SAP.

This is the valid indication:

Inject packet on path 1,2. This demonstrates the relaying facility.

```
snw ! SubnetID(0,0,0,0,0,0,0,1)
    ! Add(1, NullSNSAP)
    ! Add(1, Add(0, NullSNSAP))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PacketType(0),
                         SHF(0),
                         APID(0,0,0,0,0,0,0,0,0,0,1)),
           MakePacketSC(PacketSequenceUnSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Inject packet on path 3,1. This demonstrates relaying where the indication does not occur on the local node.

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, NullSNSAP)
    ! Add(1, Add(0, NullSNSAP))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PacketType(0),
                         SHF(0),
                         APID(0,0,0,0,0,0,0,0,0,1,1)),
           MakePacketSC(PacketSequenceUnSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,1,1),
       AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))) ;
```

Path should engage in indications in any order. This is the indication generated due to the first snw indication above:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,0,1)),
               MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

This is the local indication generated due to the first of the snw indications above:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

This is the subnetwork request generated due to the second of the snw indications above:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,1)),
               MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,1,1),
          AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))) ;

   success ; exit

)
endproc
endproc
```

Path Protocol Test9

This test checks that multicasting is possible from a path entity.

Run the test using the command:

```
ts pp.t9.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt9 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is the success event which should not be chosen if alternatives exist.

This is just a holder for the test behaviour:

```
process ppt9 (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! UserFormatted
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                           Add(1, Add(0, NullSNSAP)),
                           Add(0, NullSNSAP)),
                 Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                               Add(1, Add(0, NullSNSAP)),
                               Add(1, Add(0, Add(1, NullSNSAP)))),
                     CreateRT))) ;
```

Generate packet request on path 5,1:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Three subnetwork requests should be generated; the order is deterministic.

This is the first request:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, Add(0, Add(1, NullSNSAP)))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

This is the second request:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(0, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,1)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

This is the third request :

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,1)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   success ; exit

)
endproc
endproc
```

Path Protocol Test 10

This test checks that multicasting and routing is possible from a path entity.

Run the test using the command:

```
ts pp.t10.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt10 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be selected if alternatives exist.

This is just the holder for the test behaviour:

```
process ppt10 (pType : PacketType,
               dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,1,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! False
       ! UserFormatted
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                           Add(1, Add(0, NullSNSAP)),
                           Add(0, NullSNSAP)),
                 Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                               Add(1, Add(0, NullSNSAP)),
                               Add(1, Add(0, Add(1, NullSNSAP)))),
                     CreateRT))) ;
```

Generate incoming subnetwork indication on path 6,1:

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, NullSNSAP)
    ! Add(1, Add(0, NullSNSAP))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                          PacketType(0),
                          SHF(0),
                          APID(0,0,0,0,0,0,0,0,1,1,0)),
           MakePacketSC(PacketSequenceUnSeg,
                          PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Three subnetwork requests should be generated; the order is deterministic:

Synchronise with the subnetwork request destined for node 4:

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, Add(0, NullSNSAP))
    ! Add(1, Add(0, Add(1, NullSNSAP)))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                          PacketType(0),
                          SHF(0),
                          APID(0,0,0,0,0,0,0,0,1,1,0)),
           MakePacketSC(PacketSequenceUnSeg,
                          PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Synchronise with the subnetwork request destined for node 1:

```
snw ! SubnetID(0,0,0,0,0,0,0,1)
    ! Add(1, Add(0, NullSNSAP))
    ! Add(0, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                          PacketType(0),
                          SHF(0),
                          APID(0,0,0,0,0,0,0,0,1,1,0)),
           MakePacketSC(PacketSequenceUnSeg,
                          PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Synchronise with the subnetwork request for node 0:

```
   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   success ; exit

)
endproc
endproc
```

Path Protocol Test 11

This test attempts to send octet strings on a path which is set up as a octet source, only the valid requests should be accepted, leaving the badly formatted ones.

It tests that the secondary header flag, the sequence count, the data length, the data, the packet type and the sequence flag fields are all carried through correctly.

Run the test using the command:

```
ts pp.t11.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt11 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be chosen if there are alternatives.

This is just a holder for the test behaviour:

```
process ppt11 (pType : PacketType,
               dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw] (pType)
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType,
                                        dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] (PType : PacketType) : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! OctetString
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,0,1),
                       Add(1, Add(0, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! UserFormatted
       ! CreateRT ;
```

Attempt invalid requests:

```
((
```

A request on an incorrectly configured path:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;

   failure ; exit
)
[]
(
```

A request where the SDU exceeds the Maximum Size:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
         AddFront(Octet(0,0,0,0,0,0,1,0),
         AddFront(Octet(0,0,0,0,0,0,1,1),
         AddFront(Octet(0,0,0,0,0,1,0,0),
         AddFront(Octet(0,0,0,0,0,1,0,1),
         AddFront(Octet(0,0,0,0,0,1,1,0),
         AddFront(Octet(0,0,0,0,0,1,1,1),
         AddFront(Octet(0,0,0,0,1,0,0,0),
         AddFront(Octet(0,0,0,0,1,0,0,1),
         AddFront(Octet(0,0,0,0,1,0,1,0),
         AddFront(Octet(0,0,0,0,1,0,1,1), NullOS))))))))))) ;

   failure ; exit
)
[]
(
```

A request which is valid:

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
             Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
             Add(1, Add(0, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,0,0,1),
              AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

leading to the following subnetwork request:

```
snw ! SubnetID(0,0,0,0,0,0,0,1)
    ! Add(1, Add(0, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PType,
                         SHF(0),
                         APID(0,0,0,0,0,0,0,0,0,1,0)),
           MakePacketSC(PacketSequenceUnSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Another request with a different data length and SHF. The sequence count should have incremented.

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
             Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
             Add(1, Add(0, NullAPIDQual)))
    ! Present
    ! AddFront(Octet(0,0,0,0,0,0,1,1), NullOS) ;

snw ! SubnetID(0,0,0,0,0,0,0,1)
    ! Add(1, Add(0, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PType,
                         SHF(1),
                         APID(0,0,0,0,0,0,0,0,0,1,0)),
           MakePacketSC(PacketSequenceUnSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
       AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)) ;
```

Another request again, the sequence count should increment. Default values should have been correctly filled in as version(0,0,0), packetType(0), sequenceflag(1,1):

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
             Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
             Add(1, Add(0, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,1,0,0),
              AddFront(Octet(0,0,0,0,0,1,0,1), NullOS)) ;
```

```
snw ! SubnetID(0,0,0,0,0,0,0,1)
    ! Add(1, Add(0, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PType,
                         SHF(0),
                         APID(0,0,0,0,0,0,0,0,0,1,0)),
            MakePacketSC(PacketSequenceUnSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
            PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
         AddFront(Octet(0,0,0,0,0,1,0,0),
                   AddFront(Octet(0,0,0,0,0,1,0,1), NullOS))) ;
```

The sequence count should increment again:

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,1,1,0),
      AddFront(Octet(0,0,0,0,0,1,1,1),
      AddFront(Octet(0,0,0,0,1,0,0,0), NullOS))) ;

snw ! SubnetID(0,0,0,0,0,0,0,1)
    ! Add(1, Add(0, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PType,
                         SHF(0),
                         APID(0,0,0,0,0,0,0,0,0,1,0)),
            MakePacketSC(PacketSequenceUnSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
            PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
         AddFront(Octet(0,0,0,0,0,1,1,0),
                   AddFront(Octet(0,0,0,0,0,1,1,1),
                              AddFront(Octet(0,0,0,0,1,0,0,0), NullOS)))) ;
```

The sequence count is moving ever upwards:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
       ! Present
       ! AddFront(Octet(0,0,0,0,1,0,0,1),
         AddFront(Octet(0,0,0,0,1,0,1,0),
         AddFront(Octet(0,0,0,0,1,0,1,1),
         AddFront(Octet(0,0,0,0,1,1,0,0), NullOS)))) ;

   snw ! SubnetID(0,0,0,0,0,0,0,1)
       ! Add(1, Add(0, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PType,
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
          AddFront(Octet(0,0,0,0,1,0,0,1),
                 AddFront(Octet(0,0,0,0,1,0,1,0),
                        AddFront(Octet(0,0,0,0,1,0,1,1),
                               AddFront(Octet(0,0,0,0,1,1,0,0), NullOS))))) ;

   success ; exit
))
)
endproc
endproc
```

Path Protocol Test 12

This test injects packets into path from the subnetwork and checks that they emerge on the local node as octet strings. The main objective is to study the dataLossFlag reaction against packet repetition.

Run the test using the command:

```
ts2 pp.t12.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt12 from the menu. (As this test tries to study the dataLossFlag the parameter 'dlflag' has been assigned the value 'true'.) After that, step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is the success event which should not be chosen if there are alternatives.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This is just a holder for the test behaviour:

```
process ppt12 (PType : PacketType) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw](PType)
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (PType, true)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] (PType: PacketType) : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! true
       ! OctetString
       ! CreateRT;
```

Inject packets into path (LDP 4,1) from the subnetwork. Note that packets can come from any source SAP.

This is the valid request:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
           AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which should result in the following indication:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                  AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))
       ! OSDUNotLost;
```

Now other packet is received with seq. count increased.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
           AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which should also result in an indication:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                  AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))
       ! OSDUNotLost;
```

Eventually the same packet is received (same sequence count):

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
           AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which should also result in an indication, this time with dataLossFlag set.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))
       ! OSDULost;
```

Again correct packets are received.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,1,1),
          AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))) ;
```

Possible responses are:

A normal packet is received, the correct sequence made a reset of dataLossFlag:

```
((
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,1,1),
                   AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))
       ! OSDUNotLost;

  success; exit
)
[]
```

Although the packet received is correct, the dataLossFlag is still set.

```
(
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,1,1),
                   AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))
       ! OSDULost;

   failure ; exit
)))
endproc
endproc
```

Path Protocol Test 13

This test injects packets into path from the subnetwork and checks that they emerge on the local node as octet strings. The main objective is to study the dataLossFlag function against packet disorder.

Run the test using the command:

```
ts2 pp.t13.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt13 from the menu. (As this test tries to study the dataLossFlag the parameter 'dlflag' has been assigned the value 'true'.) After that, step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is the success event which should not be chosen if there are alternatives.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This is just a holder for the test behaviour:

```
process ppt13 (PType : PacketType) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw](PType)
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (PType, true)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] (PType: PacketType) : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! true
       ! OctetString
       ! CreateRT;
```

Inject packets into path (LDP 4,1) from the subnetwork. Note that packets can come from any source SAP.

This is the valid request:

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, NullSNSAP)
    ! Add(1, Add(0, NullSNSAP))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PacketType(0),
                         SHAbsent,
                         APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceUnSeg,
                        PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

which should result in the following indication:

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                 Add(1, NullAPIDQual))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                 Add(1, NullAPIDQual))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,0,0,1),
                AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))
    ! OSDUNotLost;
```

Now other packet is received with seq. count increased.

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, NullSNSAP)
    ! Add(1, Add(0, NullSNSAP))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PacketType(0),
                         SHAbsent,
                         APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceUnSeg,
                        PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,1,1),
       AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))) ;
```

which should also result in an indication:

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                 Add(1, NullAPIDQual))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                 Add(1, NullAPIDQual))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,0,1,1),
                AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))
    ! OSDUNotLost;
```

Eventually the packet after the next is received (sequence count increased by two):

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHAbsent,
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,1,1,1),
          AddFront(Octet(0,0,0,0,1,0,0,0), NullOS))) ;
```

which should also result in an indication, this time with dataLossFlag set.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,1,1),
         AddFront(Octet(0,0,0,0,1,0,0,0), NullOS))
       ! OSDULost;
```

Now the lazy packet is received.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHAbsent,
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,1,0,1),
          AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))) ;
```

Possible responses are:

A normal packet is received, without disorder information: dataLossFlag reset.

```
((
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,0,1),
         AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))
       ! OSDUNotLost;

  failure; stop
)
[]
```

The dataLossFlag still reflects the disorder between consecutive packets.

```
(
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,0,1),
         AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))
       ! OSDULost;
   exit
))
>>
```

The next packet is received

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,1,0,0,1),
          AddFront(Octet(0,0,0,0,1,0,1,0), NullOS))) ;
```

Possible responses are:

A normal packet is received, the disorder between the prior packet and this one is not reflected.

```
((
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,1,0,0,1),
         AddFront(Octet(0,0,0,0,1,0,1,0), NullOS))
       ! OSDUNotLost;

  failure; stop
)
[]
```

The disorder between successive packets are indicated in the dataLossFlag.

```
(
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,1,0,0,1),
         AddFront(Octet(0,0,0,0,1,0,1,0), NullOS))
       ! OSDULost;
   exit
))

>>
```

The next packet arrives :

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHAbsent,
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,1,0,1,1),
          AddFront(Octet(0,0,0,0,1,1,0,0), NullOS))) ;
```

Possible responses are:

A normal packet is received, the sequence count is the expected after the last packet.

```
((
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,1,0,1,1),
         AddFront(Octet(0,0,0,0,1,1,0,0), NullOS))
       ! OSDUNotLost;

  success; exit
)
[]
```

Although the packet received is correct, the dataLossFlag is still set.

```
(
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,1,0,1,1),
         AddFront(Octet(0,0,0,0,1,1,0,0), NullOS))
       ! OSDULost;

   failure ; exit
))

)
endproc
endproc
```

Path Protocol Test 14

This test attempts to send five octet string packets on a path which is set up as a octet string source. The goal is to measure the Path entity capability to manage repeated user requests so as to study the packet sequence count field evolution.

It tests that the secondary header flag, the sequence count, the data length, the data, the packet type and the sequence flag fields are all carried through correctly.

Run the test using the command:

```
ts2 pp.t14.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt14 from the menu. Then, step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success/failure event should only be selected when there are no alternatives.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This process holds the test behaviour:

```
process ppt14 (PType : PacketType, DLFlag : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw] (PType)
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (PType, DLFlag)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] (PType: PacketType): exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! OctetString
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                       Add(1, Add(1, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
```

Three identical octet strings are sent.

First octet string:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,0),
         AddFront(Octet(0,0,0,0,0,0,0,1),
         AddFront(Octet(0,0,0,0,0,0,1,0),
         AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)))) ;
```

Second octet string:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,0),
         AddFront(Octet(0,0,0,0,0,0,0,1),
         AddFront(Octet(0,0,0,0,0,0,1,0),
         AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)))) ;
```

(
Third Octet string:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,0),
         AddFront(Octet(0,0,0,0,0,0,0,1),
         AddFront(Octet(0,0,0,0,0,0,1,0),
         AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)))) ;
     exit
[]
     failure; stop

  ) >>
```

Two more octet strings are sent.

Fourth, different Secondary Header flag and data length:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Present
       ! AddFront(Octet(0,0,0,0,0,1,0,0),
         AddFront(Octet(0,0,0,0,0,1,0,1),
         AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))) ;
```

Fifth, different Sec. Header flag and data length:

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,1,1,1),
      AddFront(Octet(0,0,0,0,1,0,0,0),
      AddFront(Octet(0,0,0,0,1,0,0,1),
      AddFront(Octet(0,0,0,0,1,0,1,0),
      AddFront(Octet(0,0,0,0,1,0,1,1), NullOS))))) ;
```

At the subnetwork level we receive,

First packet,

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, Add(1, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PType,
                         SHAbsent,
                         APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceUnSeg,
                        PacketSequenceCount(1,1,1,1,1,1,1,1,1,1,1,1,1,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
       AddFront(Octet(0,0,0,0,0,0,0,0),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0),
       AddFront(Octet(0,0,0,0,0,0,1,1), NullOS))))) ;
```

Second packet, sequence count should be incremented

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, Add(1, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                         PType,
                         SHAbsent,
                         APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceUnSeg,
                        PacketSequenceCount(1,1,1,1,1,1,1,1,1,1,1,1,1,1)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
       AddFront(Octet(0,0,0,0,0,0,0,0),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0),
       AddFront(Octet(0,0,0,0,0,0,1,1), NullOS))))) ;
```

Third packet, sequence counts wraps around to 0:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PType,
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
          AddFront(Octet(0,0,0,0,0,0,0,0),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0),
          AddFront(Octet(0,0,0,0,0,0,1,1), NullOS))))) ;
```

Fourth subnetwork indication: different Sec. Head. Flag and data length. Sequence count continues upward.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PType,
                            SHPresent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
          AddFront(Octet(0,0,0,0,0,1,0,0),
          AddFront(Octet(0,0,0,0,0,1,0,1),
          AddFront(Octet(0,0,0,0,0,1,1,0), NullOS)))) ;
```

Fifth subnetwork indication: Sec. Header flag, data length and seq. count should have changed

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PType,
                            SHAbsent,
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
          AddFront(Octet(0,0,0,0,0,1,1,1),
          AddFront(Octet(0,0,0,0,1,0,0,0),
          AddFront(Octet(0,0,0,0,1,0,0,1),
          AddFront(Octet(0,0,0,0,1,0,1,0),
          AddFront(Octet(0,0,0,0,1,0,1,1), NullOS))))) ) ;

    success; exit
)
endproc
endproc
```

Path Protocol Test 15

This test attempts to send three user-formatted packets on a path which is set up as a packet source. The goal is to measure the path entity capacity to manage with repeated users requests.

It also tests that the secondary header flag, the sequence count, the data length, the data, the packet type and the sequence flag fields are all carried through correctly.

Run the test using the command:

```
ts2 pp.t15.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt15 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock. (Packettype parameter will have different values during this test in order to verify its fate, thus it is not requested in simulation time.)

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success/failure event should only be selected when there are no alternatives.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This process holds the test behaviour:

```
process ppt15 (DLFlag : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (PacketType(0), DLFlag)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! UserFormatted
       ! Succ(Succ(8))
       ! Add(MakeRelay(SubnetID(0,0,0,0,0,0,1,0),
                       Add(1, Add(1, NullSNSAP)),
                       Add(1, NullSNSAP)),
             CreateRT) ;
```

Three identical packets are sent.

First packet:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Second packet:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
(
hide success, failure in
```

Third packet and subsequent indications:

```
(  pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PacketType(0),
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Fourth Packet: It has different parameter values to show that relevant details are carried through correctly.

```
pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
               Add(1, Add(0, NullAPIDQual)))
    ! Add(1, Add(0, NullAPIDQual))
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                          PacketType(1),
                          SHF(1),
                          APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceFirstSeg,
                         PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
       AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;
```

At the subnetwork level we receive,

First packet,

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, Add(1, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                          PacketType(0),
                          SHF(0),
                          APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceUnSeg,
                          PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Second subnetwork indication:

```
snw ! SubnetID(0,0,0,0,0,0,1,0)
    ! Add(1, Add(1, NullSNSAP))
    ! Add(1, NullSNSAP)
    ! MakeCCSDSPacket
        (MakePrimaryHeader
           (MakePacketID(Version1,
                          PacketType(0),
                          SHF(0),
                          APID(0,0,0,0,0,0,0,0,1,0,0)),
           MakePacketSC(PacketSequenceUnSeg,
                          PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
           PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
       AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Third subnetwork indication:

```
  snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Fourth subnetwork indication:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceFirstSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;

   success ; exit
)
[]
(
```

Or, the entity is locked

```
 failure ; exit
)
)
endproc
endproc
```

Path Protocol Test 16

This test attempts to receive three packets from the subnetwork. But the user could not receive them as they arrived (because i.e., a bit overhead). The goal is to measure the path entity capacity to manage with repeated subnetworks indications.

Run the test using the command:

```
ts2 pp.t16.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt16 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success/failure event should only be selected when there are no alternatives.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This process holds the test behaviour:

```
process ppt16 (PType : PacketType, DLFlag : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw](PType)
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (PType, DLFlag)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] (PType: PacketType): exit :=
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! true
       ! UserFormatted
       ! CreateRT;
```

At the subnetwork level we receive,

First packet,

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PType,
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
               MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
           AddFront(Octet(0,0,0,0,0,0,0,1),
           AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Second subnetwork indication:

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PType,
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
               MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
           AddFront(Octet(0,0,0,0,0,0,0,1),
           AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

```
(
 hide success, failure in
(
```

Third subnetwork indication:

```
  snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(1, NullSNSAP))
       ! Add(1, NullSNSAP)
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PType,
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
               MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
               PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
           AddFront(Octet(0,0,0,0,0,0,0,1),
           AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Finally the user begins to attend the Path Entity.

First packet:

```
  pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PType,
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Second packet:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PType,
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Third packet:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                   Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                            PType,
                            SHF(0),
                            APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   success ; exit
)
[]
```

```
(
```

Or, the entity is locked:

```
 failure ; exit
)
)

endproc
endproc
```

<u>Path Service Test 1</u>

This test attempts to send an octet string and a packet on a path which has not been set up as a source.

Run the test using the command:

```
ts ps.t1.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst1 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be chosen unless no alternatives exist

This is just a holder for the test behaviour:

```
process pst1 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(

(
```

Attempt to send an octet string on a path which has not yet been set up. Should be rejected.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,0),
                   AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;

   failure ; exit
)
[]
(
```

Attempt to send a user formatted packet on the same path. Reject:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,0,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,0,1)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,0),
             AddFront(Octet(0,0,0,0,0,0,0,1), NullOS))) ;
   failure ; exit
)
[]
(
```

This should be the only possible event:

```
   success ; exit
)
)

endproc
endproc
```

Path Service Test 2

This test sets up an octet source and a user formatted source and then attempts to send a user formatted packet on the path set up as an octet source and an octet string on the path set up as a user formatted source.

Run the test using the command:

```
ts ps.t2.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst2 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is with the success event which should not be chosen if alternatives are offered.

This process just holds the behaviour

```
process pst2 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! OctetString
       ! Add(MakeEndpoint(OctetString, True), CreateTL)
       ! PacketType(0)
       ! Succ(Succ(8)) ;

   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(1, NullAPIDQual)))
       ! UserFormatted
       ! Add(MakeEndpoint(OctetString, True), CreateTL)
       ! PacketType(0)
       ! Succ(Succ(8)) ;
((
```

Attempt to send a user-formatted packet on an octet string path; rejected:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
             AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
 )
[]
(
```

Attempt to send an octet string on a user formatted path; rejected:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(1, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(1, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,0),
                   AddFront(Octet(0,0,0,0,0,0,0,1), NullOS)) ;

   failure ; exit
)
[]
 (
   success ; exit
))
)
endproc
endproc
```

Path Service Test 3

This test attempts to send user-formatted packets on a path which is set up as a packet source, only the valid packet should be accepted, leaving the badly formatted ones.

It tests that the secondary header flag, the sequence count, the data length, the data, the packet type and the sequence flag fields are all carried through correctly.

Run the test using the command:

```
ts ps.t3.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst3 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test, the exceptions to this are that the success event which should only be chosen if there are no alternatives, and that the losepacket event should not be chosen at all.

This is just a holder for the behaviour

```
process pst3 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the path through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                      Add(1, Add(0, NullAPIDQual)))
       ! UserFormatted
       ! Add(MakeEndpoint(UserFormatted, True), CreateTL)
       ! PacketType(0)
       ! Succ(Succ(8)) ;
```

Attempt to send a badly user-formatted packet on a path set up as a packet source.

```
(
(
```

A packet where the APID is not a user APID:

```
   pkt ! MakePathID(APID(1,1,1,1,1,1,1,1,1,1,1),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(1,1,1,1,1,1,1,1,1,1,1)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the version is not version 1:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version(0,0,1),
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet where the packet length is not consistent with data:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which is valid, for this test allow all packets to travel through space link in order, with none missing (i.e., do not use the losepacket event):

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

resulting in the following indication:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Another packet request with different SHF, sequence count, data length and user data. The default fields (version and packetType) must have their correct values for the packet to be accepted.

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)) ;
```

resulting in the following indication:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)) ;

   success ; exit
))
)
endproc
endproc
```

Path Service Test 3a

This test attempts to send user-formatted packets on a path which is set up as a packet source, only the valid packet should be accepted, leaving the badly formatted ones.

Run the test using the command:

```
ts2 ps.t3a.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst3a from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test, the exceptions to this are that the success event which should only be chosen if there are no alternatives, and that the losepacket event should not be chosen at all.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This process holds the test behaviour:

```
process pst3a : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! UserFormatted
       ! Add(MakeEndpoint(OctetString, true),
         CreateTL)
       ! Packettype(0)
       ! succ(succ(8));
```

Attempt to send a badly user-formatted packet on a path set up as a packet source.

```
((
```

A packet where SDU length exceeds the maximum set by manager.

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0),
          AddFront(Octet(0,0,0,0,0,0,1,1),
          AddFront(Octet(0,0,0,0,0,1,0,0),
          AddFront(Octet(0,0,0,0,0,1,0,1), NullOS))))) ) ;

   failure ; exit
)
[]
(
   success ; exit
)
)
)
endproc
endproc
```

Path Service Test 4

This test checks that multicasting is possible from a path entity using the path service.

A path will be set up which has three endpoints.

Run the test using the command:

```
ts ps.t4.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst4 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test, the exceptions are that the losepacket event should not be chosen at all and that the success event should only be selected if there are no other choices.

This is just a holder for the test behaviour:

```
process pst4 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the path through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! UserFormatted
       ! Add(MakeEndpoint(OctetString, False),
         Add(MakeEndpoint(UserFormatted, False),
         Add(MakeEndpoint(UserFormatted, False), CreateTL)))
       ! PacketType(0)
       ! Succ(Succ(8)) ;
```

Generate packet request on path 5,1:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,1,0,1)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Three indications should be generated, the order is non-deterministic. No packets should be lost during this test (i.e., do not use the losepacket event).

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;

   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,1,0,1)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,1,0,1)),
                MakePacketSC(PacketSequenceUnSeg,
                              PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
                       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
   success ; exit

)
endproc
endproc
```

Path Service Test 5

This test attempts to send octet strings on a path which is set up as a octet source, only the valid requests should be accepted, leaving the badly formatted ones.

It tests that the secondary header flag, the sequence count, the data length, the data, the packet type and the sequence flag fields are all carried through correctly.

Run the test using the command:

```
ts ps.t5.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst5 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test. Exceptions to this are the losepacket event which should only be selected at the time indicated below in the test spec. and the success event which should only be selected when there are no other alternatives.

This process just looks after the test behaviour

```
process pst5 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! OctetString
       ! Add(MakeEndpoint(OctetString, True), CreateTL)
       ! PacketType(0)
       ! Succ(Succ(8)) ;
```

Attempt invalid requests:

```
((
```

A request on an unconfigured path:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;

   failure ; exit
)
[]
(
```

A request which is valid:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,0,0,1),
                   AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

And the indication produced at the other end of the path:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       !  AddFront(Octet(0,0,0,0,0,0,0,1),
                    AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))
       ! OSDUNotLost ;
```

Another request with a different SHF and data:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Present
       ! AddFront(Octet(0,0,0,0,0,0,1,1), NullOS) ;
```

And another indication, again with the data loss flag:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Present
       ! AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)
       ! OSDUNotLost ;
```

Another request, this time the sequence count wraps. This checks that the service keeps track of sequence counts correctly.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,0,0),
                   AddFront(Octet(0,0,0,0,0,1,0,1), NullOS)) ;
```

Another indication with data loss flag:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,0,0),
          AddFront(Octet(0,0,0,0,0,1,0,1), NullOS))
       ! OSDUNotLost ;
```

Yet another request; this one should be lost by selecting the losepacket event when offered with a packet containing the user data in this request:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,1,0),
         AddFront(Octet(0,0,0,0,0,1,1,1),
         AddFront(Octet(0,0,0,0,1,0,0,0), NullOS))) ;
```

This request should be allowed through, i.e., do not select losepacket with this user data.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Present
       ! AddFront(Octet(0,0,0,0,1,0,0,1),
         AddFront(Octet(0,0,0,0,1,0,1,0),
         AddFront(Octet(0,0,0,0,1,0,1,1),
         AddFront(Octet(0,0,0,0,1,1,0,0), NullOS)))) ;
```

This indication will have the data loss flag set as the packet before should have been lost.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Present
       ! AddFront(Octet(0,0,0,0,1,0,0,1),
          AddFront(Octet(0,0,0,0,1,0,1,0),
           AddFront(Octet(0,0,0,0,1,0,1,1),
            AddFront(Octet(0,0,0,0,1,1,0,0), NullOS))))
       ! OSDULost ;
```

The indicator should now have re-synch'ed with the request above, the data loss flag should be reset in the octet indication.

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,0,0),
                   AddFront(Octet(0,0,0,0,0,1,0,1), NullOS)) ;
```

So the indicator shows no packet missing:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Absent
       ! AddFront(Octet(0,0,0,0,0,1,0,0),
                   AddFront(Octet(0,0,0,0,0,1,0,1), NullOS))
       ! OSDUNotLost ;

   success ; exit
))
)
endproc
endproc
```

Path Service Test 5a

This test injects invalid subnetwork.indications into a path entity from the subnetwork and checks that they are disallowed.

Run the test using the command:

```
ts pp.t5a.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select ppt5a from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should not be selected unless there is no alternative.

This process is just a holder for the test behaviour:

```
process ppt5a (pType : PacketType,
              dataLossFlags : Bool) : noexit :=
(
   hide pkt, oct, man, snw in
   (
      test[pkt, oct, man, snw]
      |[pkt, oct, man, snw]|
      PathProtocol[pkt, oct, man, snw] (pType, dataLossFlags)
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man,snw] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,0),
                    Add(1, NullAPIDQual))
       ! Succ(Succ(8))
       ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, Add(0, NullSNSAP))
       ! True
       ! UserFormatted
       ! CreateRT ;
```

Inject packets into path from the subnetwork on LDP 4,1:

```
(
(
```

A packet where the APID has not been set up by management. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, NullSNSAP))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
```

A packet which exceeds the MaxSDU parameter. This should be rejected.

```
   snw ! SubnetID(0,0,0,0,0,0,1,0)
       ! Add(1, NullSNSAP)
       ! Add(1, Add(0, Add(0, Add(1, Add(0, NullSNSAP)))))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,1,0,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;

   failure ; exit
)
[]
(
   success ; exit
)
)
)
endproc
endproc
```

Path Service Test 6

This test attempts to send user-formatted packets on a path which is set up as a packet source. It demonstrates multicasting to endpoints offering different services and that the subnetwork can lose requests.

Run the test using the command:

```
ts ps.t6.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst6 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this rule is that the losepacket event should only be selected when offered with the packet specified in the test below and that the success event should only be selected when there are no other alternatives.

This process just holds the test behaviour:

```
process pst6 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This process gives the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the path through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! UserFormatted
       ! Add(MakeEndpoint(UserFormatted, False),
         Add(MakeEndpoint(OctetString, False), CreateTL))
       ! PacketType(0)
       ! Succ(Succ(8)) ;
```

A packet which is valid, for this test allow all packets to travel through space link in order:

```
pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                 Add(1, NullAPIDQual))
    ! Add(1, NullAPIDQual)
    ! MakeCCSDSPacket
         (MakePrimaryHeader
            (MakePacketID(Version1,
                           PacketType(0),
                           SHF(0),
                           APID(0,0,0,0,0,0,0,0,1,0,1)),
             MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
                    AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Packet indication at node 1:

```
pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                 Add(1, NullAPIDQual))
    ! Add(1, NullAPIDQual)
    ! MakeCCSDSPacket
         (MakePrimaryHeader
            (MakePacketID(Version1,
                           PacketType(0),
                           SHF(0),
                           APID(0,0,0,0,0,0,0,0,1,0,1)),
             MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
                    AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Octet indication at node 3:

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                 Add(1, NullAPIDQual))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                 Add(1, NullAPIDQual))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,0,0,1),
                AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

Lose this packet on the packet channel, get it through to octet channel. This is done by selecting the losepacket event when offered with the packet given here.

```
pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                 Add(1, NullAPIDQual))
    ! Add(1, NullAPIDQual)
    ! MakeCCSDSPacket
         (MakePrimaryHeader
            (MakePacketID(Version1,
                           PacketType(1),
                           SHF(1),
                           APID(0,0,0,0,0,0,0,0,1,0,1)),
             MakePacketSC(PacketSequenceUnSeg,
                           PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,0,1,1), NullOS)) ;
```

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Present
       ! AddFront(Octet(0,0,0,0,0,0,1,1), NullOS) ;
```

Send this packet through:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,1,0,1)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             AddFront(Octet(0,0,0,0,0,1,0,0), NullOS)) ;
```

Resulting in the following indications:

```
   oct ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Present
       ! AddFront(Octet(0,0,0,0,0,1,0,0), NullOS) ;

   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,1,0,1),
                    Add(1, NullAPIDQual))
       ! Add(1, NullAPIDQual)
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                             PacketType(1),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,1,0,1)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,1,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
             AddFront(Octet(0,0,0,0,0,1,0,0), NullOS)) ;

   success ; exit
)
endproc
endproc
```

Path Service Test 7

This test attempts to send four octet strings on a path which is set up as a octet source before any indication may be received.
The goal is to check the Path Service against multiple requests and study the packet sequence control evolution.

Run the test using the command:

```
ts2 ps.t7.lot ps.lot
```

Type START at the hippo prompt (after some time) and then select pst7 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test. Exceptions to this are the losepacket event which should only be selected at the time indicated below in the test spec. and the success event which should only be selected when there are no other alternatives.

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This process just looks after the test behaviour:

```
process pst7 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour itself:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the paths through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                  Add(1, Add(0, NullAPIDQual)))
       ! OctetString
       ! Add(MakeEndpoint(UserFormatted, true),
         CreateTL)
       ! Packettype(0)
       ! succ(succ(8));
```

First request.

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,0,0,1),
      AddFront(Octet(0,0,0,0,0,0,1,0), NullOS)) ;
```

Second request.

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! Present
    ! AddFront(Octet(0,0,0,0,0,0,1,1),
      AddFront(Octet(0,0,0,0,0,1,0,0),
      AddFront(Octet(0,0,0,0,0,1,0,1),
      AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))));
```

Third request.

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(1, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,0,1,1,1), NullOS);
```

Fourth request.

```
oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
               Add(1, Add(0, NullAPIDQual)))
    ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
               Add(1, Add(0, NullAPIDQual)))
    ! Absent
    ! AddFront(Octet(0,0,0,0,1,0,0,0),
      AddFront(Octet(0,0,0,0,1,0,0,1),
      AddFront(Octet(0,0,0,0,1,0,1,0),
      AddFront(Octet(0,0,0,0,1,0,1,1), NullOS))));
```

leading to the following indications:

```
((
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                  Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(1,1,1,1,1,1,1,1,1,1,1,1,1,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
          AddFront(Octet(0,0,0,0,0,0,0,1),
          AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
    exit
|||
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                  Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(1),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(1,1,1,1,1,1,1,1,1,1,1,1,1,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
          AddFront(Octet(0,0,0,0,0,0,1,1),
          AddFront(Octet(0,0,0,0,0,1,0,0),
          AddFront(Octet(0,0,0,0,0,1,0,1),
          AddFront(Octet(0,0,0,0,0,1,1,0), NullOS)))) );
    exit
|||
 pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                  Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                            PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
          AddFront(Octet(0,0,0,0,0,1,1,1), NullOS));
   exit
|||
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                  Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
           (MakePrimaryHeader
              (MakePacketID(Version1,
                             PacketType(0),
                             SHF(0),
                             APID(0,0,0,0,0,0,0,0,0,1,0)),
              MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
              PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1)),
           AddFront(Octet(0,0,0,0,1,0,0,0),
           AddFront(Octet(0,0,0,0,1,0,0,1),
           AddFront(Octet(0,0,0,0,1,0,1,0),
           AddFront(Octet(0,0,0,0,1,0,1,1), NullOS)))) );
   exit
) >> success; exit )
[]
(  failure ; exit )
)

endproc
endproc
```

Path Service Test 8

This test tries to send three user-formatted packets before any indication may be received. The goal is to know if the Path service can cope with repeated requests and, in that case, if disordering in received indications is possible. The success even means only that multiple requests can be managed but the disorder issue should be tested inspecting succeeding data received.

Run the test using the command:

```
ts2 ps.t8.lot pp.lot
```

Type START at the hippo prompt (after some time) and then select pst8 from the menu. Step through the events offered using the NEXT command of hippo. Eventually the success event should take place and the next NEXT command will result in deadlock.

The choice taken when multiple events are offered is not important for the success of this test; the exception to this is that the success event should only be selected when there are no alternatives

The test result is successful only if the 'success' event happens (deadlock or failures before this event mean the specification does not accomplish the test).

This process just holds the behaviour

```
process pst8 : noexit :=
(
   hide pkt, oct, man in
   (
      test[pkt, oct, man]
      |[pkt, oct, man]|
      PathService[pkt, oct, man]
   )
)

where
```

This is the test behaviour:

```
process test[pkt,oct,man] : exit :=

hide success, failure in
(
```

Set up the path through the management gate:

```
   man ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! UserFormatted
       ! Add(MakeEndpoint(OctetString, true),
         CreateTL)
       ! Packettype(0)
       ! succ(succ(8));
```

First packet:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,0,1),
             AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))) ;
```

Second packet request:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,0,1,1),
             AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))) ;
```

Third packet request:

```
   pkt ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                    Add(1, Add(0, NullAPIDQual)))
       ! Add(1, Add(0, NullAPIDQual))
       ! MakeCCSDSPacket
            (MakePrimaryHeader
               (MakePacketID(Version1,
                              PacketType(0),
                              SHF(0),
                              APID(0,0,0,0,0,0,0,0,0,1,0)),
                MakePacketSC(PacketSequenceUnSeg,
                             PacketSequenceCount(0,0,0,0,0,0,0,0,0,0,0,0,0,0)),
                PacketLength(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)),
             AddFront(Octet(0,0,0,0,0,1,0,1),
             AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))) ;

((
(
```

First indication:

```
 oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
     ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                 Add(1, Add(0, NullAPIDQual)))
     ! Absent
     ! AddFront(Octet(0,0,0,0,0,0,0,1),
       AddFront(Octet(0,0,0,0,0,0,1,0), NullOS))
     ! OSDUNotLost;
     exit
|||
```

Second indication:

```
 oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                   Add(1, Add(0, NullAPIDQual)))
     ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                   Add(1, Add(0, NullAPIDQual)))
     ! Absent
     ! AddFront(Octet(0,0,0,0,0,0,1,1),
       AddFront(Octet(0,0,0,0,0,1,0,0), NullOS))
     ! OSDUNotLost;
     exit
|||
```

And, third indication:

```
 oct ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                   Add(1, Add(0, NullAPIDQual)))
     ! MakePathID(APID(0,0,0,0,0,0,0,0,0,1,0),
                   Add(1, Add(0, NullAPIDQual)))
     ! Absent
     ! AddFront(Octet(0,0,0,0,0,1,0,1),
       AddFront(Octet(0,0,0,0,0,1,1,0), NullOS))
     ! OSDUNotLost;
     exit
) >> success; exit
)
[]
(
   failure; exit
)
)
)
endproc
endproc
```